

Freakduino Long Range Wireless Board WalkThrough - Part 2

[| Print |](#)

Written by Akiba

Friday, 25 October 2013

In the [first part of the walkthrough](#), we learned some basic operations and hello world type programs to get the 900 LR board up and running. In this walkthrough, we'll be building on what we learned previously and moving on to more advanced topics like radio configuration and power management.

Adding Commands to the Command Line

In the last section of the walkthrough, part 1, I introduced the [cmdArduino command line library](#). It allows you to make sketches interactive from the serial terminal. Now, let's expand on that a bit and add custom commands that we can call from the command line.

First, we're going to write the command function that we'll be calling from the command line. The function needs to be in a specific format but otherwise, the actual functionality is left up to you. The format is as follows:

```
void funcName(int argCnt, char **args)
{
}
```

Don't worry if it's unintelligible. Here's an explanation of what happens. When you type in the command at the command line, you can also type in arguments, ie: "setsaddr 3" would set the short address to the value of 3. The command line library parses the command "setsaddr 3" into command line arguments, in this case 2 arguments. The first argument is always the command name, "setsaddr" and has an argument index of 0. The second argument is the ASCII value "3" and has an argument index of 1. The variable "args" is a string array which separates every string written on the command line by their spaces and stores them. The variable argCnt contains the total number of arguments typed in at the command line.

Let's make an actual command function that we'll call from the command line. This function will echo any command entered into the command line:

```
void cmdEcho(int argCnt, char **args)
{
    int i;
    for (i=0; i<argCnt; i++)
    {
        Serial.print("Arg ");
        Serial.print(i);
        Serial.print(" = ");
        Serial.print(args[i]);
        Serial.println();
    }
}
```

In this function, we loop through all the arguments, print out their index number, and then print out the actual argument entered into the command line. We used argCnt to figure out how many arguments we need to print out, and then printed the argument directly from the array. We now need to link this function to a command on the command line. We do this in the setup() function.

In the setup function, we can link commands to the command line using the "chibiCmdAdd()" library call. In this case, to link our echo function, we'd add this line of code:

```
chibiCmdAdd("echo", cmdEcho);
```

Now, when we type "echo" and then a series of arguments, the command line parser will call the function "cmdEcho" and pass in the arguments and the argument count. Let's try that out:

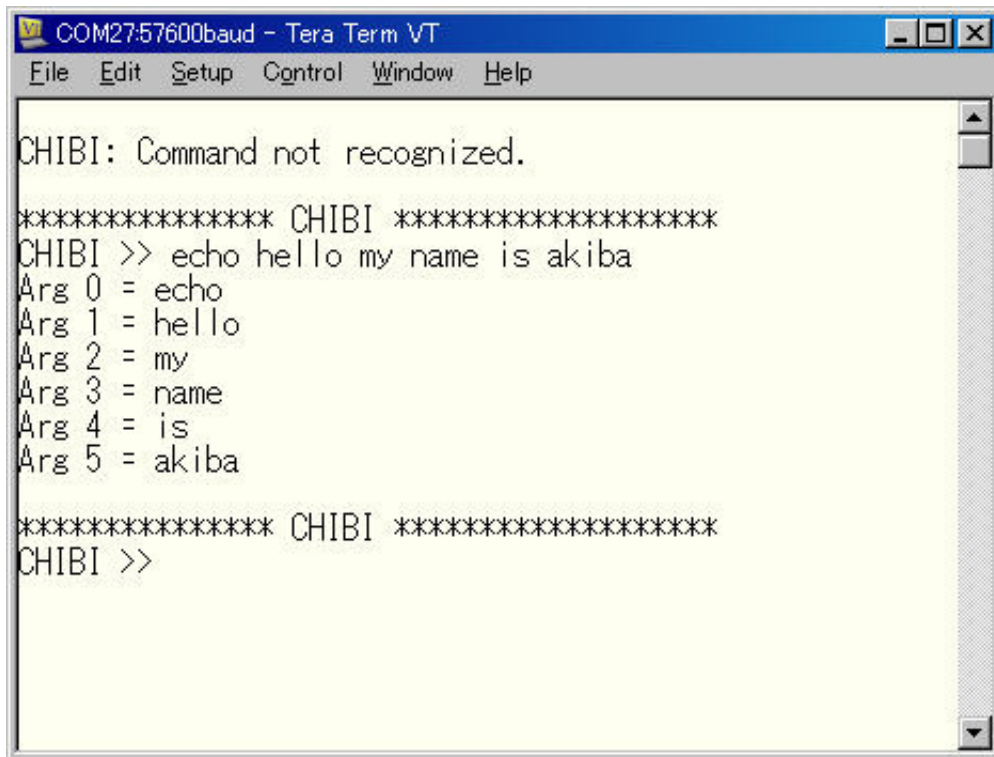
```
#include <chibi.h>

void setup()
{
    chibiCmdInit(57600);
    chibiCmdAdd("echo", cmdEcho);
}

void loop()
{
    chibiCmdPoll();
}

void cmdEcho(int argCnt, char **args)
{
    int i;
    for (i=0; i<argCnt; i++)
    {
        Serial.print("Arg ");
        Serial.print(i);
        Serial.print(" = ");
        Serial.print(args[i]);
        Serial.println();
    }
}
```

Here's what the output looks like from a serial terminal:

A screenshot of a Tera Term VT serial terminal window. The title bar reads "COM27:57600baud - Tera Term VT". The menu bar includes "File", "Edit", "Setup", "Control", "Window", and "Help". The terminal output shows a prompt "CHIBI:" followed by "Command not recognized." and then a separator line "***** CHIBI *****". Below this, the user enters "CHIBI >> echo hello my name is akiba". The terminal then displays the arguments: "Arg 0 = echo", "Arg 1 = hello", "Arg 2 = my", "Arg 3 = name", "Arg 4 = is", and "Arg 5 = akiba". Another separator line "***** CHIBI *****" follows, and the prompt "CHIBI >>" is shown at the bottom.

```
COM27:57600baud - Tera Term VT
File Edit Setup Control Window Help

CHIBI: Command not recognized.

***** CHIBI *****
CHIBI >> echo hello my name is akiba
Arg 0 = echo
Arg 1 = hello
Arg 2 = my
Arg 3 = name
Arg 4 = is
Arg 5 = akiba

***** CHIBI *****
CHIBI >>
```

You can see that each string I wrote on the command line is chopped up and stored in the args array. We can now access all the arguments written on the command line and process them however we need to.

Finally, it's common to want to pass in numerical arguments rather than string arguments. Since the command line only parses strings, we'll need to convert from an ASCII string into an integer inside our command function. To do this, there's a special function called "chibiCmdStr2Num()". The

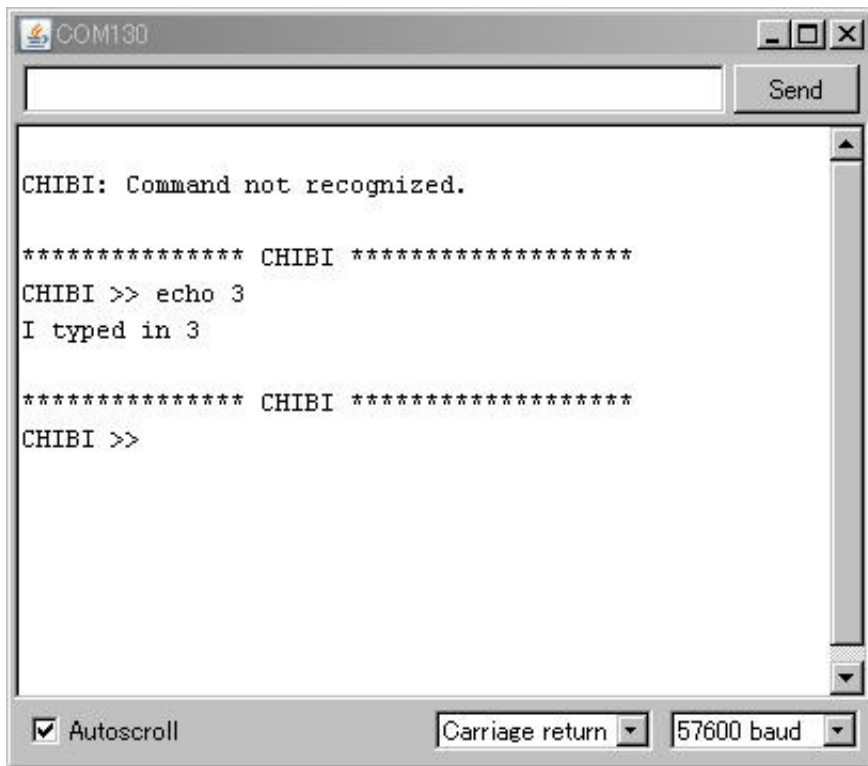
format is:

```
chibiCmdStr2Num("input string", base);
```

where the input string is the numerical ASCII string you want to convert to a number and the base is the numerical base you want to use. For decimal, the base should be 10. For hexadecimal, the base should be 16. Let's try modifying the cmdEcho function to print out a number:

```
void cmdEcho(int argCnt, char **args)
{
    int val;
    val = chibiCmdStr2Num(args[1], 10);
    Serial.print("I typed in ");
    Serial.println(val);
}
```

Now, if you run this sketch and type "echo 3", you should see it print out I typed in 3". Here's what it looks like from a serial terminal:



Now that you can control your sketches interactively from the command line, I hope you find this part of the chibiArduino library useful in your development :)

Adding printf to the Sketch

It's often much more concise to use printf to print to the serial port rather than Serial.print() because you can combine variable data and text in one printf statement. For example, in the last example, we printed out the numerical argument that we typed at the command line. To do this, we had to do two separate Serial.print statements. If we had printf, we could have combined it into one statement:

```
printf("I typed in: %d\n", val);
```

If you're not familiar with this syntax, then you can google how to use printf, but it makes printing things to the serial port much more convenient.

It's especially useful for debugging and is one of the principal ways software devs debug kernel drivers. This is how you can enable printf functionality in your Arduino sketch:

```
static FILE uartout = {0};

void setup()
{
    // fill in the UART file descriptor with pointer to writer.
    fdev_setup_stream (&uartout, uart_putchar, NULL, _FDEV_SETUP_WRITE);

    // The uart is the standard output device STDOUT.
    stdout = &uartout ;
}

void loop()
{
}

/*****
// This is to implement the printf function from within arduino
*****/
static int uart_putchar (char c, FILE *stream)
{
    Serial.write(c);
    return 0;
}
```

The discussion of what's going on behind the scenes is a bit esoteric. Basically, you need to set up a buffer for the serial output (which is of type FILE). This is called a stream and you need to redirect serial output to that stream. When the printf() function is called, it will send the text data to that stream and will then call the function passed into the second argument of fdev_setup_stream(). For that function, we need to implement something that will print out a single character. There are many ways to do this, but we'll just do a simple Serial.write(). That's it. After adding those three lines of code and the one function, you now have printf() capability.

Radio Configuration

For the Freakduino 900 LR board, radio configuration is important. Understanding how to change basic functions of the radio will improve the functionality of the board immensely.

Changing the channel

Changing the channel is such a common operation that I implemented a function just to handle that. Each standard channel has a different center frequency. It's important to know which frequency to operate on based on your location.

The default channel frequencies are divided into 11 channels with center frequencies as follows:

Channel: Center Frequency

- 0: 868.3 MHz
- 1: 906 MHz
- 2: 908 MHz
- 3: 910 MHz
- 4: 912 MHz
- 5: 914 MHz
- 6: 916 MHz
- 7: 918 MHz
- 8: 920 MHz
- 9: 922 MHz
- 10: 924 MHz

To select one of these channels, you can just use the following function:

```
chibiSetChannel(channelNum);
```

In North America, it's possible to operate up to 1W in the 902 to 928 MHz bands for spread spectrum systems. In Europe, the regulations are slightly different. It's possible to operate up to 500 mW at 869.4 MHz at a 10% duty cycle or using Listen Before Talk. This means that the radio needs to be configured for 869.4 MHz which is not in the standard channel set.

The AT86RF212 radio used on the 900 LR board also allows setting frequencies from 857.0 to 882.5 MHz with 0.1 MHz channel spacing. This means that it's possible to set the radio for 869.4 MHz. To do this, you'll need to set the registers in the radio directly. There are two registers that need to be set. The first register is called CC_CTRL_1 (address 0x14). The bottom three bits set the radio frequency band. For the 857 to 882.5 MHz band, we need to set this to 2:

```
chibiRegWrite(0x14, 0x02);
```

Once the band is set, we need to tune it to 869.4 MHz. The formula is:

Frequency = 857.0 MHz + (0.1 * CC_NUMBER)

This means that for 869.4 MHz, our CC_NUMBER value needs to be 124 (0x7C). In the CC_NUMBER register (address 0x13), we need to write 0x7C:

```
chibiRegWrite(0x13, 0x7C);
```

The final code we have for setting our frequency to 869.4 MHz should look like this

```
#include <chibi.h>
void setup()
{
    chibiInit();
    chibiRegWrite(0x14, 0x02); // set freq band
    chibiRegWrite(0x13, 0x7C); // set cc_number
}
```

For further information on the topic of setting the frequencies, especially to nonstandard frequencies, you should check the [Atmel AT86RF212 datasheet](#), section 7.8, Frequency Synthesizer Section.

Setting the Modulation

One of the main things people are interested in for wireless sensor applications is range. For WSN, we rarely care about high data rates but range is usually important. The AT86RF212 radio on the 900 LR board supports two different types of modulation: OQPSK and BPSK. Without getting into too many details regarding digital modulation, OQPSK is a more complex modulation but allows higher data rates, up to 1 Mbps. The default modulation is OQPSK at 250 kbps which is standardized by IEEE 802.15.4-2006.

It's also possible to increase the range of the radio by decreasing the modulation complexity. BPSK is a much simpler modulation scheme which will allow the receiver to discriminate digital values at much lower receiver sensitivities. This translates into longer range communications. The tradeoff is that the maximum data rate is 40 kbps due to the simpler modulation, i.e.: less bits per symbol. To change the modulation, you can use the following function call:

```
chibiSetMode(mode);
```

The following are valid modes:

```
OQPSK_SINRC
OQPSK_SIN
OQPSK_RC
BPSK_40
```

The default modulation is OQPSK_SIN. It's best to use either the default or BPSK_40 for longer range. Here's how to set the radio for BPSK_40:

```
chibiSetMode(BPSK_40);
```

For the descriptions of the other modulations, it's best to consult the AT86RF212 datasheet, section 7.1, Physical Layer Modes.

Increasing the DataRate

It's also possible to increase the data rate of the Freakduino 900 LR board. The maximum data rate at 900 MHz is 1 Mbps for payload data. The headers will still be sent at 250 kbps so the total average datarate will be less than the maximum specified data rate (unless its 250 kbps). To change the datarate, you'd use this library call:

```
chibiSetDataRate(rate);
```

The valid values for arguments for the long range board are:

```
CHB_RATE_250KBPS  
CHB_RATE_500KBPS  
CHB_RATE_1000KBPS
```

The default is CHB_RATE_250KBPS and its possible to go up to 1000 Kbps. This may be useful for some applications that require high data rates like audio or media transmission, or high resolution transmissions such as seismic data.

Power Management

Power management is probably one of the most important aspects of wireless sensor networks, especially for environmental monitoring. The assumption is that the device will be completely wireless and hence autonomous. There's even a special name for this called "resource constrained networks". The Freakduino 900 LR has been designed for low power operation and can get down to around 180 uA at the battery in sleep mode. At these levels it's possible to survive over a year on 2 standard AA batteries.

A fairly standard power management strategy would be to keep the device asleep as long as possible, wake up at pre-determined intervals to measure data, transmit the data, and then go back to sleep. Putting the board to sleep is a sequential process. In active mode, the board consumes about 140 mA in high gain mode where the Rx amplifier is enabled or 45 mA if the Rx amplifier is not enabled.

Before you begin, please check to make sure you have at least [v1.04 of the chibiArduino library](#). There were some modifications to the code to make it easier to shut down the radio.



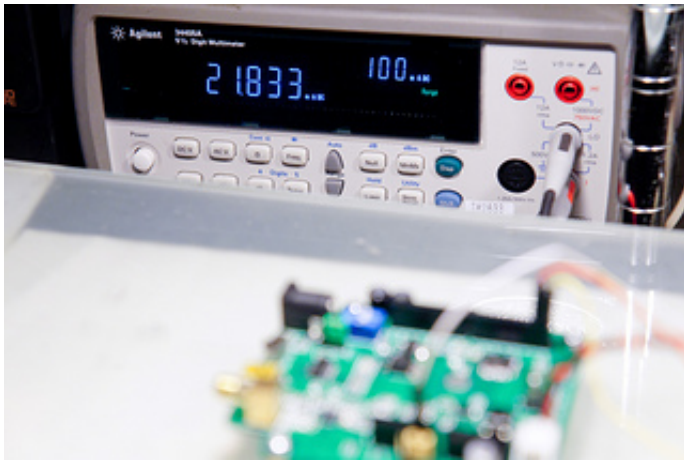
To reduce power, one of the first things that needs to be done is to put the radio to sleep. This is easily done using the following function.

```
chibiSleepRadio(enable);
```

Passing in an argument of 1 or true will put the radio to sleep. Passing in a value of 0 or false will wake the radio back up and put it into operational mode. Once the radio is put to sleep, the SPI pins and radio control pins need to be set low and forced to inputs. To do this, we're going to take a shortcut and manipulate the MCU's control registers directly. In practice, its best to save off these values so they can be restored when the device wakes back up.

```
PORTB = 0;  
DDRB = 0;
```

With the radio asleep, the power consumption of the bare board should drop to about 22 mA:



After the radio is put to sleep, and the control pins are set to low and input, the next thing to do is to put the MCU to sleep. Putting the MCU to sleep is also sequential where various peripheral blocks need to be shut down. Once the active peripherals are shut down, the MCU can be put into a deep sleep mode. To do this, we're going to need to use libraries outside of the core Arduino libraries. Before you continue, you'll need to access some special AVR libraries from avrlibc which will need to be included:

```
#include <avr/sleep.h>  
#include <avr/power.h>
```

Once again, it's good practice to save off the register values in a variable before clearing them. When the MCU goes through the wakeup process, it will be much easier to restore them to their active state.

The first action in our MCU power down sequence is to disable the UART. This will prevent the MCU from continually driving values to the USB serial chip and also prevent the USB serial chip from leeching from the MCU. To do this, we set the UART register to 0x00:

```
// disable UART  
UCSR0B = 0x00;
```

Next, we set all the remaining GPIO ports low and then set them to inputs:

```
PORTC = 0x00;  
PORTD = 0x00;  
DDRC = 0x00;  
DDRD = 0x00;
```

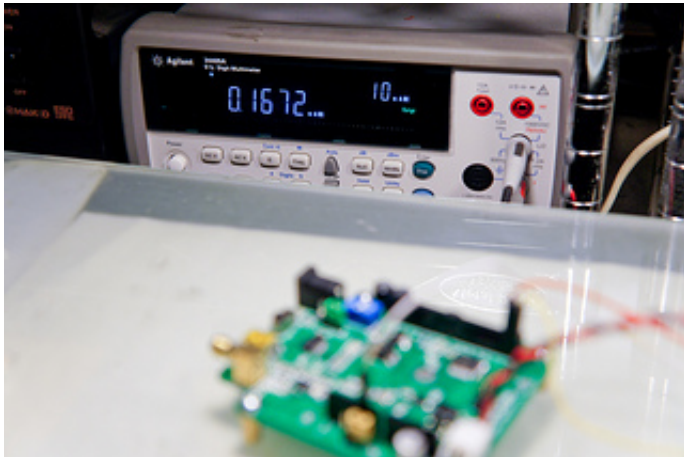

Next we disable the ADC (analog digital converter) which is enabled for the analogRead functions:

```
ADCSRA &= ~(1 << ADEN);    // Disable ADC
```

The final steps are to put the MCU into power down mode which is the lowest power sleep mode available. Once this is done, the MCU will not be accessible until it is woken up.

```
set_sleep_mode(SLEEP_MODE_PWR_DOWN);
sleep_enable();
sleep_mode();
```

Once the board is fully shut down, the power consumption can drop below 180 uA. At this point, the main power consumers are the boost converters (2 x ~60 uA) and the quiescent current of the voltage regulator (~60 uA). The only way to wake the MCU up is from a reset, an external interrupt, watchdog timeout, or an I2C access. Care should be taken when thinking about the power management strategy to figure out how the MCU will be woken up.



That's it for this walkthrough. We went through some fairly advanced topics in embedded design and sensor networks. I hope you found it interesting and useful and if you haven't already, please check out the [freakduino series of boards available in the shop](#) . As you can see, they're quite versatile, especially for wireless communications and wireless sensor networks.

Hits: 79962

[Email This](#)

Trackback(0)

 [TrackBack URI for this entry](#)

Comments (3)

 [Subscribe to this comment's feed](#)

Archaic?

written by [MRE](#), October 25, 2013

In the discussion about using Printf, I think you mean to use the word esoteric, not archaic.

Votes: +0

[vote up](#)

[vote down](#)

[report abuse](#)



...

written by [MRE](#), October 25, 2013



Lots of good info about how best to cut the current during sleep by the way. Thanks. I am sure I am missing a lot of these helpful hints.
Bookmarked.

Votes: +0

vote up

vote down

report abuse

...

written by Akiba, October 25, 2013

You say tomato, I say red juicy thang. But yeah, I changed it to esoteric. Perhaps I've been in Japan too long.

Votes: +1

vote up

vote down

report abuse



Write comment



Show/Hide comment form

[Close Window](#)